

NORTHWEST NAZARENE UNIVERSITY

Four-Band Image Acquisition System

THESIS


Submitted to the Department of Mathematics and Computer Science
in partial fulfillment of the requirements
for the degree of
BACHELOR OF SCIENCE

Peter R. Oxley
2017


THESIS
Submitted to the Department of Mathematics and Computer Science
in partial fulfillment of the requirements
for the degree of
BACHELOR OF SCIENCE

Peter R. Oxley
2017


Four-Band Image Acquisition System

Author: 


Peter Oxley

Approved: 

Dale Hamilton, M.S., Assistant Professor of Computer Science
Department of Mathematics and Computer Science, Faculty Advisor

Approved: 

Julie Straight, Ph.D., Associate Professor of English
Department of Language, Literature, and Cultural Studies, Second Reader

Approved: 

Barry Myers, Ph.D., Chair
Department of Mathematics and Computer Science

Abstract

Four-Band Image Acquisition System.

OXLEY, PETER (Department of Math and Computer Science), HAMILTON, DALE
(Department of Math and Computer Science).

An imaging system for acquiring electromagnetic reflectance data was specified, designed, assembled, and tested. The acquisition device captures reflectance in the 400-1000 nanometer wavelength spectrum, which is divided into three visible-light bands (red, green, and blue) and the near-infrared band. The acquired data can be manipulated in several ways to generate information about plant health, moisture content, and genus. The system is built on a Raspberry Pi 3B and uses dual imaging sensors as well as a number of position, motion, and orientation sensors, which allow the device to precisely locate the captured images. The system is designed to be mounted on an unmanned aerial vehicle in order to acquire a series of over-head images, which can later be stitched together in an orthomosaic. Onboard device-control software was also developed and tested, allowing each capture sequence to be customized with user options, which can be set locally or remotely via an integrated user interface. This system is designed such that it can be reproduced using components and production processes that are readily available to most universities, and at a fraction of the cost of comparable commercially-available systems.

Acknowledgments

This research was supported in part by funding from NNU FireMAP. The author wishes to thank the members of this Thesis Committee, the Computer Science Faculty Advisors, the members of the FireMAP Research Team, Esteban Cano for patiently teaching the author to solder, and all those who shared their knowledge and experience in the form of documentation and online forum postings.

Table of Contents

Abstract	iii
Acknowledgments.....	iv
Table of Figures	vi
Four-Band Image Acquisition System.....	1
Application of Near-Infrared Data.....	2
System Specification.....	3
System Design and Setup.....	5
Development Board	5
Imaging Sensors.....	5
Location and Orientation Sensors	6
Software Development.....	10
User Interface.....	10
Configuration File.....	11
Device Software.....	11
Results.....	14
Future Development.....	15
Conclusion	16
References.....	18
Appendices.....	19
Appendix A: Website References	19
Appendix B: Parts List and Sources	20
Appendix C: User Interface	21
Appendix D: Configuration File	23
Appendix E: Device Control.....	25

Table of Figures

Figure 1. Visual Spectrum and Near-Infrared Band (Resonant FM, 2016).....	2
Figure 2. Triangulation Problem.....	7
Figure 3. FourEyes User Interface	10
Figure 4. Earliest Opportunity Model Flowchart.....	13
Figure 5. RGB and NIR+GB compared.....	14
Figure 6. Sample Log Data	15

Four-Band Image Acquisition System

Until just a few years ago, unmanned flight was almost exclusively limited to expensive “remote controlled” aircraft that had to be continuously controlled by a ground-based pilot. However, recent advances in small-scale unmanned flight mean that many unmanned aircraft are relatively inexpensive, are capable of semi-autonomous travel along a pre-programmed flight path, and are able to carry a small payload. These developments have created a variety of opportunities for hobbyists, businesses, researchers, and others to gather information from an aerial perspective. At the same time, advancements in computer and sensor technology have made it possible to develop small, light-weight, inexpensive devices intended for a specific application. While most manufacturers of unmanned aircraft offer a camera that can take still images or video, many owners are taking advantage of the airborne platform to carry other devices and sensors, many of which are custom designed for the user’s needs.

One such specific application involves using multi-spectral imaging to evaluate plant health and moisture content. It has been shown that the amount of near-infrared light that a plant reflects is indicative of the plant health and moisture content (Rouse, 1974). Many consumer-grade digital cameras can be modified to capture images that include near-infrared data, and these modified cameras have become valuable agricultural tools used for monitoring crop health. NNU’s Fire Monitoring and Assessment Platform (FireMAP) research team became convinced that this same technology could be applied to assessing moisture content and health of wildland plants, and that the information generated could inform wildland fire management.

Application of Near-Infrared Data

Electromagnetic radiation exists in a wide range of wavelengths. The narrow range that humans are able to perceive visually is between approximately 400 and 700 nanometers and is often subdivided into three bands: red, green, and blue (RGB).

Radiation of longer wavelengths, between 700 and one million nanometers, is classified as infrared (IR). The narrow subsection of the IR spectrum from approximately 700 to 1000 nanometers is usually described as near-infrared (NIR). Figure 1. Visual Spectrum and Near-Infrared Band illustrates the various bands of electromagnetic radiation

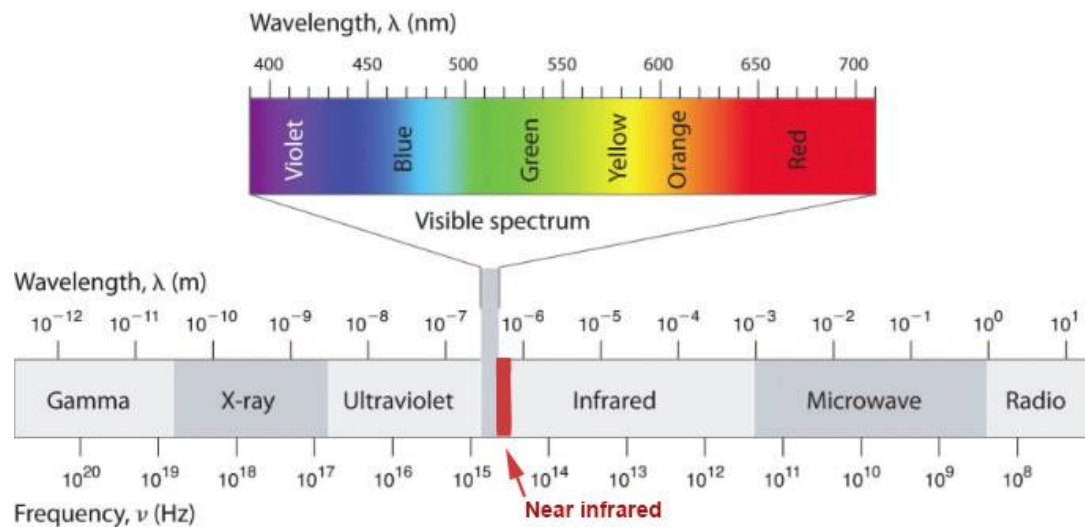


Figure 1. Visual Spectrum and Near-Infrared Band (Resonant FM, 2016)

(Resonant FM, 2016).

Healthy plant leaves contain high amounts of chlorophyll, which absorbs a significant amount of red and blue light for the photosynthesis process. As health declines, the amount of chlorophyll that is present decreases as well, and more visual spectrum light is reflected. Conversely, the cell structure of viable leaves reflects a high percentage of NIR light, but drier leaves absorb more light from the NIR band (Weier & Herring, 2000). Simply measuring the amount of NIR reflectance can give a limited

amount of information about plant health, but much better information is found by comparing NIR and red reflectance. A standard formula called the Normalized Difference Vegetation Index (NDVI) is used to express the ratio of reflectance in these two ranges (Holden, 2010). NDVI is calculated using $NDVI = (NIR - red)/(NIR + red)$. The NDVI calculation produces a number between negative one (-1) and positive one (+1). A result near zero indicates that no viable plant material is present, while a result near +1 indicates healthy plant material (Weier & Herring, 2000). Each pixel in an image can be evaluated for NDVI, or blocks of pixels may be averaged together.

System Specification

As mentioned previously, many consumer-grade digital cameras can be modified to capture NIR data. Inexpensive kits are available to make the conversion (Public Lab, 2016). Initially, NNU FireMAP intended to simply use modified cameras mounted on small unmanned aerial vehicles (UAV) to capture NIR imagery. A camera was modified, but early testing identified several drawbacks to this approach. Among the most significant are the following:

1. Consumer-grade cameras are designed to capture three-band images. Modifying them to capture NIR means sacrificing one of the other bands (usually red). This can significantly reduce the accuracy the NDVI calculation since it depends on comparing the red and NIR values. Later research revealed that analysis of the RGB bands allows for accurate categorization of vegetation types, making the loss of data from one band even more undesirable.
2. Consumer-grade cameras of the type and quality that are appropriate for modification tend to be heavy, approaching the maximum payload capacity of the

UAVs in use by FireMAP.

3. Because the FireMAP system is intended to survey large areas of wildland, the captured imagery must include reasonably accurate location information. Some consumer-grade cameras offer internal Global Positioning System (GPS) sensors, but these cameras are heavier and significantly more expensive than their counterparts without GPS functionality.
4. No consumer-grade camera was found that could be programmed to capture an appropriately-timed sequence of photos.

The FireMAP team also determined that using a professional-grade multi-band sensor was not an option because of the weight and cost of available models.

After exploring other options, the FireMAP team decided to specify a custom device to capture the desired data. The project was nicknamed “FourEyes,” and the initial specifications included the following device requirements:

1. Inexpensive (relative to existing devices): The device must represent a cost savings relative to readily-available sensors. Four-band devices that can capture reflectance data in the desired spectra are available for approximately \$5000. Reproductions of this project should be no more than 10% of that cost.
2. Airframe agnostic: The device must be designed to be usable on virtually any airframe configuration.
3. Light weight: The device must not hinder the flight or maneuverability of the UAV to which it is attached. The airframes that are expected to carry this device are in the 350mm, 1.5kg range.
4. Replicable by FireMAP or its partners: Multiple devices may be desired for

simultaneous data acquisition and to replace any that are damaged or lost. The device must be designed in such a way as to be manufactured and calibrated with minimal difficulty.

System Design and Setup

Development Board

The development board was the first component selected. All the other components had to be compatible with the development board, and it had to be sufficient to fulfill the FourEyes design requirements. The Raspberry Pi 3B was selected for several reasons, but a large determining factor was the presence of a camera port and the availability of inexpensive, compatible imaging sensors. Other factors included the low price-point, the wide variety of compatible sensors, built-in Wi-Fi, the flexibility of the operating system, and the active and enthusiastic user community. The only significant disadvantage to the Raspberry Pi 3B was the weight. Other models of Raspberry Pi were found to be lighter, but each sacrificed some valuable feature.

Imaging Sensors

The Raspberry Pi Foundation offered cameras specifically designed for compatibility with the Raspberry Pi devices. At nearly the same time as this project was started, the Foundation began to offer an NIR camera. While the choice of imaging sensors was obvious, the Raspberry Pi boards only included a single port for attaching a camera, and no reasonable option for connecting a second camera. The option to use a USB camera for the second imaging device was rejected because of the weight and the external connection that would be required. The Raspberry Pi Compute Module IO Board was considered, as it has two camera ports, but this option was eventually rejected

because the IO Board is significantly larger, heavier, and more expensive than the Raspberry Pi 3B. Further research uncovered a small company in Turkey called IVMech that was manufacturing a device designed specifically for attaching multiple cameras to a Raspberry Pi. The IVMech IVPort device was a “multiplexer,” which powered two or more cameras simultaneously, while rapidly switching between the data signals from each camera. The first IVPort device was ordered from IVMech in early July, 2016, just before the failed *coup d'état* in Turkey. As a result, the supplier was unable to make the shipment for several days, but remained in regular contact, and when the device finally arrived, the supplier provided excellent support via email. However, after a few exchanges, email responses from IVMech suddenly ceased and the English version of the IVMech website became unavailable. The website remains unavailable as of this date (March, 2017). (It is the author’s sincere hope that the employees of IVMech and their family members are safe). Later in 2016, a similar multiplexer device, which is manufactured in China, became available on Amazon.

Location and Orientation Sensors

The FireMAP system requires images to be located and overlaid on a map in a format called an orthomosaic. This requires relatively accurate information about the location where a photo was taken. Sufficient accuracy is provided by most Global Positioning System (GPS) devices. However, further complication is introduced when capturing imagery from a UAV, because the imaging sensors may not be pointed straight down. As Figure 2 illustrates, this means that the location of the device and the location of the image center could be somewhat different. This introduces the need for some way to measure the tilt and orientation of the imaging sensors. GPS devices offer some

orientation information, and a device called an “accelerometer” can provide tilt information.

GPS and accelerometer devices each have limitations. GPS orientation is determined by direction of motion, rather than relation to Magnetic North. This means that if the device is facing a different direction than it is travelling, it will erroneously report that it is facing in the direction of travel. An accelerometer measures

the acceleration forces to which it is exposed. If it is traveling at a constant speed and direction, then the data it produces can be used to calculate the angle of tilt, although these calculations are quite complicated. Unfortunately, UAV flight rarely involves constant speed and direction, and any course corrections, jostling from wind currents, or elevation adjustments will reduce the accuracy of the tilt measurements. The result is that conditions that improve the accuracy of one device will reduce the accuracy of the other. Despite these limitations, pairing the GPS and accelerometer sensors seemed to be the best way to gather data, and it was hoped that the image location could be calculated to an acceptable level of precision.

The NNU Computer Science Department had recently purchased GPS and

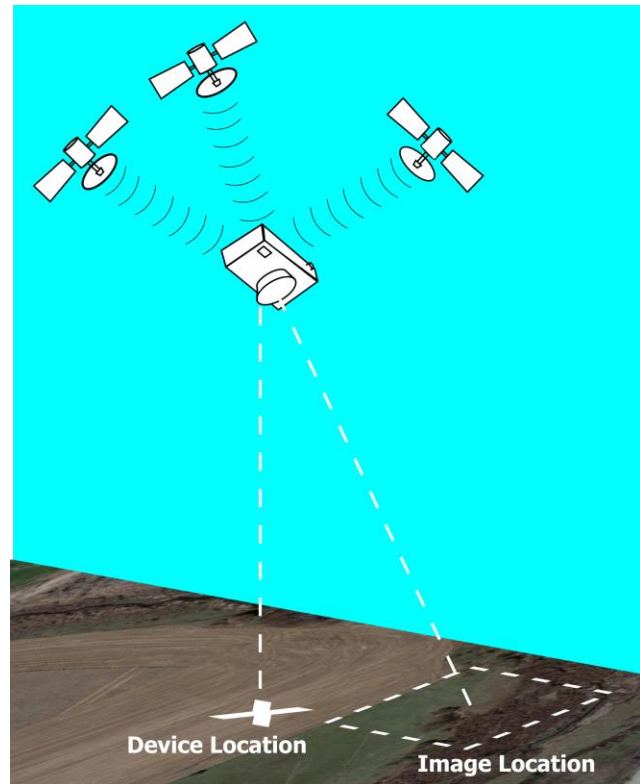


Figure 2. Triangulation Problem

accelerometer devices manufactured by Microstack, so these were appropriated for this project. Unfortunately, these were found to be difficult to configure for the Raspberry Pi development board. The Microstack devices were designed to be compatible with previous versions of Raspberry Pi, and updated configuration information was not readily available. Adafruit Industries offered similar devices that were specified as being compatible with the Raspberry Pi 3B. These were ordered and successfully connected following the documentation provided by Adafruit.

The difficulty in calculating tilt in three dimensions (3D) was much more significant than anticipated, but the NNU Mathematics Department generously offered help and support. Before this solution had been pursued for very long, Professor Dale Hamilton was told of a device called an “absolute orientation board” (AOB). This device has an array of sensors and performs several calculations on-board, allowing it to capture orientation to Magnetic North and the 3D angle of tilt, among other interesting data. Replacing the accelerometer with the AOB would remove the need to calculate 3D orientation in software and would overcome the shortcomings of the GPS orientation. An AOB was purchased from Adafruit.

The AOB introduced a new challenge in the development of the project as it needed to be connected to the same general purpose input/output (GPIO) pins as the GPS device. Adafruit offered a USB adapter that works with the GPS board, but this was unacceptable because using the adapter would require including an external cable. No other alternative was found, so a switching device was proposed. This device would utilize a small microcontroller to rapidly switch between the GPS and AOB devices, similar to the function of the camera multiplexer. Unfortunately, the project developer

had no prior experience with embedded systems, so preparing a microcontroller to serve this function required a large, unexpected investment of time. A kit supplied by Adafruit was assembled to facilitate programming a microcontroller, and an adapter was designed and assembled for use with the selected microcontroller. When development of the microcontroller solution was nearly complete, a suitable adapter board was found that, with some modification, could be installed without external cables. In the interest of time, and to minimize the difficulty of replicating the project, the microcontroller solution was abandoned and the adapter was specified.

Because the Raspberry Pi is actually a small computer, it must be shut down before disconnecting the power supply, which means that simply switching a battery can require approximately 60 seconds for the system to shut down and restart. When this happens, power to the GPS sensor is also lost and upon restart, the GPS device will often take 90 seconds or more to reacquire a sufficient signal. Two-and-a-half minutes is an unacceptably long time requirement for simply switching batteries, so a second power port was added to allow hot-swapping of power supplies.

Connections between hardware components were accomplished by means of soldering or using jumper-wire connections. While these connectors add a small amount of weight and take up additional space, they allow for easy replacement or rearrangement of components, and they facilitate temporary disconnection for troubleshooting purposes.

At this point, all the specified components had been installed and were working together. Through some experimentation, a suitable case design was established, and a model created using RhinoCAD drafting software. The case was then fabricated using one of NNU's 3D printers.

Software Development

The programming language selected for developing the device control software was Python 3. The user interface (UI) is simply a local webpage written in the ubiquitous Hypertext Markup Language (HTML), and the configuration file is an Extensible Markup Language (XML) document. Taken together, these three files are a very simple application of the Model-View-Controller (MVC) software architecture. The current version of the FourEyes code (March, 2017) can be found in the appendices.

User Interface

The UI is the user's access point to the control software: the "view" in the MVC design scheme. It is intended to offer the user a simple way to adjust the device options and trigger the data capture sequence. Using a local webpage for the UI offers several

advantages. Most importantly, a webpage will be supported by almost any device, allowing users to control the system with whatever device they choose. Another consideration is the ease of developing a webpage as compared to writing an application to perform the same function. Additionally, web-style interfaces will be

FourEyes User Preferences

Start delay:

- altitude: Delay sequence until n meters above START
- distance: Delay sequence until n meters away from START
- time: Delay sequence until n seconds after START

n =

Capture trigger:

- distance: Capture image every n meters
- speed: Capture image when UAV drops below n meters per second
- time: Capture image every n seconds

n =

Termination condition:

- altitude: Terminate sequence when UAV returns to within n meters above START
- frames: Terminate sequence after capturing n frames
- time: Terminate sequence after n seconds

n =

Maximum rotational velocity:

degrees: Delay frame if UAV is rotating faster than n degrees per second

n =

Log frequency:

time: Record and log data every n seconds

n =

Directory:

location: Save log and images

START

Figure 3. FourEyes User Interface

familiar to most users, allowing them to understand how to interact with the UI without a lot of prompts or training (see Figure 3. FourEyes User Interface).

When the UI loads, it accesses the last-used settings, which are stored in the configuration file. Any changes the user makes are written to the configuration file and used as instructions for the device software.

Configuration File

The configuration file stores the adjustable parameters of the system. It is the “controller” in the MVC architecture. Most of the settings stored here are the user preferences, which are adjusted using the UI. However, any data that needs to be stored by the device software between flights would be written here as well. When the user triggers the data capture sequence, the device software looks up the control parameters in the configuration file.

Using the XML format for the configuration file offers a couple of advantages. Many modern programming languages offer libraries or programming interfaces to access and to write to XML documents, making the configuration file easily accessible to both the UI and the device software. Also, a major feature of XML is that it is “extensible.” In simple terms, this means that changes and improvements can be made to an XML document without necessarily requiring any change to the programs that access the file. This is useful because it allows features to be added and tested without modifying the whole system.

Device Software

The device software does the work of triggering the various sensors and aggregating the data they return. Perhaps a little confusingly, the software that controls

the device is the “model” portion of the MVC framework. When the user triggers the data capture sequence, the device software looks up the appropriate settings in the configuration file and uses them to schedule the sequence.

This software was written in Python primarily because Python is a convenient language for development. It is broadly accepted and supported by a large user base, and most of its syntax and conventions are easy to use and read. Python is also a language that is widely used by the Raspberry Pi user base. Conveniently, Python happens to be the language in which the Adafruit device interfaces are written, which greatly simplifies the interaction with the device software.

The main drawback to using Python is that Python code executes more slowly than many others because it is evaluated as it runs, rather than being a compiled language. For device control, the slightly slower execution can create issues when trying to access multiple sub-systems as nearly simultaneously as possible. This issue is most noticeable when trying to switch quickly between the two imaging sensors.

The device software must complete two different repetitive tasks: data logging and image acquisition. These happen on different schedules, and the frequency of image acquisition must be allowed to vary even within a single acquisition flight without significantly impacting the frequency of data logging. For instance, under certain conditions, an image capture could be delayed for several seconds, but logging should continue. In this system, the user may choose to capture images based on distance or speed, rather than time, so it is not appropriate to use a strict timer-model to trigger data logging and image acquisition. Instead, FourEyes uses an “earliest opportunity” model for data capture. The earliest opportunity model, illustrated in Figure 4. Earliest

Opportunity Model Flowchart, defines milestones that, when passed, give permission to a data capture event. On completion of that event, a new milestone is defined. If a milestone has not been reached for a certain event, or if the capture event must be postponed, the software proceeds to evaluate milestones for other events and retries the initial event again at the next opportunity. In this way, the data is not captured on a strict schedule, but at the earliest appropriate opportunity. This provides significant flexibility to the data capture process.

The primary reason that image capture might be delayed is high angular velocity

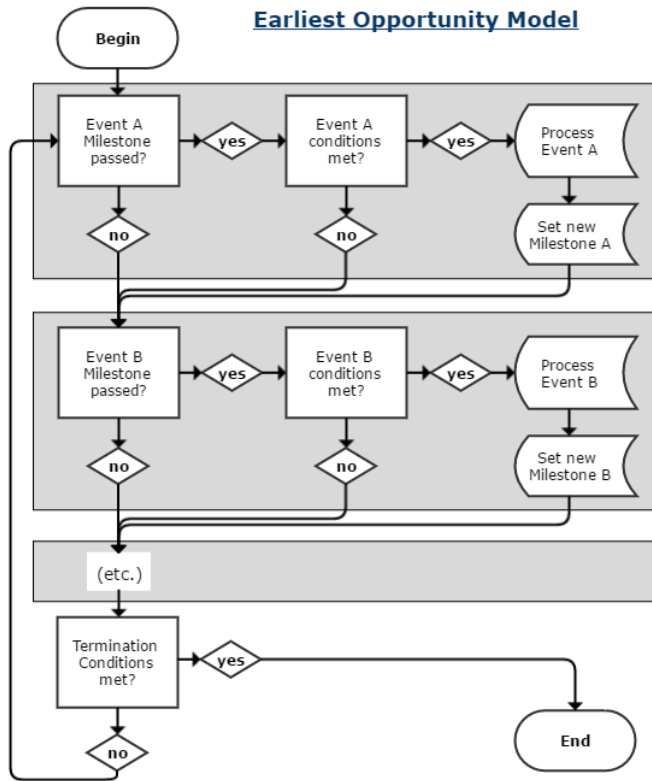


Figure 4. Earliest Opportunity Model Flowchart

(rotational speed) of the device.

The system attempts to take two images as nearly simultaneously as possible, but a slight delay in both the software and the hardware create a brief interval between the capture of each image. If the UAV is turning during an image capture sequence, the two images will not align well. Fortunately, this project adopted the AOB, which calculates angular velocity. If an

image capture milestone is reached, but high angular velocity is detected, the image capture is postponed.

For each data log milestone, flight data is captured. Some of this data provides a record of the flight path and flight conditions; other data is used to determine if a milestone has been reached. When an image capture milestone is reached, and when flight and device conditions allow, the RGB and NIR sensors are triggered and the images are captured and saved to file. At that point an additional record is entered in the log to indicate the location and conditions when the photos were taken.

Results

After an acquisition flight is complete, the images and flight data are uploaded to the FireMAP system, and the location and orientation data are used to overlay the images

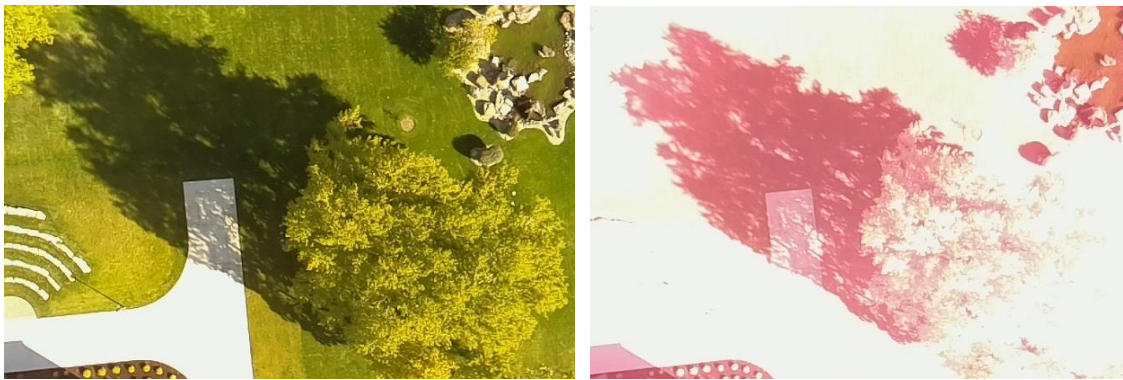


Figure 5. RGB and NIR+GB compared

accurately on a map in an orthomosaic. Of the two images captured by FourEyes, one is RGB and the other is NIR+GB. A pair of images is shown in Figure 5. RGB and NIR+GB compared. The NIR from one image can be compared to the red from the other image for true NDVI calculation. Since both images have the green and blue bands, this data can be compared to ensure accurate alignment of the two images. Once aligned, FireMAP has the option to treat each of the four reflectance bands as a separate map layer, or to convert the image into a single four-band image. Images from multiple acquisition flights can be “stitched” together to represent large areas of contiguous data.

In addition to helping locate the images, the data collected in the logfile can provide FireMAP information about the conditions in which the images were captured. The data in the logfile can also help FireMAP analyze and flight anomalies and provide an accurate record of the flightpath. A sample data log is shown in Figure 6.

time,	img,	lat,	lon,	alt,	agl,	climb,	speed,	sats,	epx,	epy,	epv,	track,	head,	roll,	pitch,	<v
13:27:44,	-,	43.562947,	-116.565993,	864.8,	97.9,	0.73,	4.9,	7,	3,	3,	9,	89.7,	92.7,	3.2,	5.2,	3.1
13:27:49,	-,	43.562840,	-116.565993,	866.1,	99.2,	0.32,	5.1,	7,	3,	3,	9,	89.9,	91.9,	5.2,	5.5,	5.8
13:27:50,	1,	43.562810,	-116.565991,	866.3,	99.4,	0.09,	4.8,	7,	3,	3,	9,	90.1,	92.0,	5.7,	5.3,	-0.9
13:27:54,	-,	43.562813,	-116.565891,	866.8,	99.9,	0.10,	4.9,	7,	3,	3,	9,	90.1,	91.7,	4.9,	5.7,	4.3
13:27:59,	-,	43.562807,	-116.565736,	866.5,	99.6,	-0.07,	5.2,	7,	3,	3,	9,	90.3,	92.2,	5.1,	5.3,	-6.2
13:28:01,	2,	43.562810,	-116.565524,	866.7,	99.8,	0.00,	5.0,	7,	3,	3,	9,	89.8,	92.3,	5.7,	5.2,	-3.3
13:28:05,	-,	43.562815,	-116.565355,	866.8,	99.9,	0.02,	4.9,	8,	3,	3,	7,	90.0,	91.1,	5.4,	5.6,	4.4
13:28:10,	-,	43.562818,	-116.565183,	865.9,	99.0,	-0.41,	5.4,	8,	3,	3,	7,	89.9,	91.6,	7.4,	5.5,	4.1
13:28:11,	3,	43.562818,	-116.565143,	866.4,	99.5,	0.77,	4.9,	8,	3,	3,	7,	90.0,	91.3,	5.7,	5.5,	2.5

Figure 6. Sample Log Data

Future Development

There are several areas in which the FourEyes system could be improved, refined, or extended. On the hardware side, the device could be made lighter. One option for doing this is to modify the Raspberry Pi 3B board itself to remove unnecessary physical components. Another option to explore is the new Raspberry Pi Zero W, which, unlike the previous version, has Wi-Fi on-board. Improving the sensor-wire connector system could slightly reduce weight, but more importantly, it would make the cabling more organized and easier to connect and disconnect when needed.

On the software side, there are several possible improvements as well. Rewriting the device software in the C programming language should help it to execute more quickly, possibly reducing the delay between capture of image pairs. Another improvement that could reduce the delay would be to buffer the first image in RAM while taking the second photo. Currently, the system must finish saving the first image to disk before the second image is captured, a process that introduces a delay of approximately 0.2 seconds.

In the current system design, the coordinate information for the images is only stored in the data log. While it is a relatively simple operation to parse that data when creating an orthomosaic, it might be worthwhile to have those data imbedded in the metadata header of the photos so that the image files would still be useful if they were ever separated from the data log.

At present, the UI design has not been optimized for hand-held devices. While the UI webpage should be compatible with tablets and smartphones, the user may have to resize the webpage to interact with it. Adding Cascading Style Sheet (CSS) functionality to the View could enable the UI to adjust dynamically to any screen size.

The current system configuration requires FourEyes and the user's device to be connected to the same network via a router. This requires a portable router of some sort to be brought to the acquisition site. However, a Raspberry Pi is capable of being configured to serve as a router. Adding this feature to FourEyes would mean one less device was required on-site. After moving out of range, the FourEyes device occasionally has difficulty reconnecting to the router when it returns. This problem would be eliminated if the FourEyes device were acting as the system router.

Conclusion

The FourEyes project was very challenging, and a significant amount of time was spent developing subsystems that were later abandoned. While individual components were well documented, combining them in this way created several new difficulties. However, the final product was a system that met all the initial project specifications and provided a detailed record of a data acquisition flight and pairs of images that can be analyzed for information that is valuable to the FireMAP project. The project should be

replicable by anyone with reasonable soldering skills and access to a 3D printer, and for a total cost of under \$300, which was significantly less than commercially-available options. Further development of this project could produce an image acquisition system that is even more useful and user-friendly.

References

- Holden, Z. A. (2010). Beyond Landsat: A Comparison of Four Satellite Sensors for Detecting Burn Severity in Ponderosa Pine Forests of the Gila Wilderness, NM, USA. *International Journal of Wildland Fire*, 449-458.
- Public Lab. (2016). *Infragram DIY Filter Pack*. Retrieved from Public Lab: <https://publiclab.myshopify.com/collections/diy-infrared-photography/products/infragram-diy-filter-pack>
- Resonant FM. (2016, March 28). *Red and Near-Infrared Light*. Retrieved from Resonant FM: <http://www.resonantfm.com/red-and-near-infrared-light-buyers-guide-part-i-incandescent-halogen-and-heat-bulbs>
- Rouse, J. H. (1974). Monitoring Vegetation Systems in the Great Plains with ERTS. *Proceedings, 3rd Earth Resource Technology Satellited (ERTS) Symposium, 1*, 46-62.
- Weier, J., & Herring, D. (2000, August 30). *Measuring Vegetation: NDVI & EVI*. Retrieved from Earth Observatory: <https://earthobservatory.nasa.gov/Features/MeasuringVegetation>

Appendices

Appendix A: Website References

Adafruit: www.adafruit.com

Amazon: www.amazon.com

IVMech: www.ivmech.com

Microstack: www.microstack.org.uk

Python: www.python.org

Raspberry Pi Foundation: www.raspberrypi.org

RhinoCAD: www.rhino3d.com

Appendix B: Parts List and Sources

Absolute Orientation Board: Adafruit. www.adafruit.com/product/2472

Device Case: Printable using file at <http://github.com/poxley/4eyes>

Multiplexer: Amazon: <http://a.co/fXk9AWR>

Raspberry Pi 3B: Adafruit. www.adafruit.com/product/3055

Raspberry Pi Camera Rev 1.3: Amazon. <http://a.co/7bwhFnv>

Raspberry Pi NoIR Camera Rev 1.3: Amazon. <http://a.co/aA09s46>

(NOTE: v2.1 cameras not compatible with multiplexors)

Ultimate GPS Board: Adafruit: www.adafruit.com/product/746

USB Micro-B Board: Adafruit: www.adafruit.com/product/1833

USB to Serial Adapter: Amazon: <http://a.co/a1FWGcN>

Also required are generic jumpers, headers, and miscellaneous components

Hardware Options to Explore

Arducam Milti Camera Adapter Board: Amazon. <http://a.co/3sqh2dj>

Raspberry Pi 3B component reduction: lifehacker. <https://goo.gl/sUkGwM>

Raspberry Pi Zero W: Adafruit. www.adafruit.com/product/3400

I2C to UART Bridge: Sandbox Electronics. <https://goo.gl/ss9vZG>

Appendix C: User Interface

More recent version may be available at <http://github.com/poxley/4eyes>

```
<!DOCTYPE html>
<!-- 4eyes User Interface -->
<html>
<h1>FourEyes User Preferences</h1>
  <form name="UI" method="POST" action="">

    <fieldset>
      <legend>Start delay:</legend>
      <input type="radio" name="delay" value="altitude" checked>
altitude: Delay sequence until n meters above START<br>
      <input type="radio" name="delay" value="distance">
distance: Delay sequence until n meters away from START<br>
      <input type="radio" name="delay" value="time"> time: Delay
sequence until n seconds after START<br>
      n = <input type="number" name="delayValue">
    </fieldset>

    <fieldset>
      <legend>Capture trigger:</legend>
      <input type="radio" name="trigger" value="distance"
checked> distance: Capture image every n meters<br>
      <input type="radio" name="trigger" value="speed"> speed:
Capture image when UAV drops below n meters per second<br>
      <input type="radio" name="trigger" value="time"> time:
Capture image every n seconds<br>
      n = <input type="number" name="triggerValue">
    </fieldset>

    <fieldset>
      <legend>Termination condition:</legend>
      <input type="radio" name="termination" value="altitude"
checked> altitude: Terminate sequence when UAV returns to within n
meters above START<br>
      <input type="radio" name="termination" value="frames">
frames: Terminate sequence after capturing n frames<br>
      <input type="radio" name="termination" value="time"> time:
Termiate sequence after n seconds<br>
      n = <input type="number" name="terminateValue">
    </fieldset>

    <fieldset>
      <legend>Maximum rotational velocity:</legend>
degrees: Delay frame if UAV is rotating faster than n
degrees per second<br>
      n = <input type="number" name="rotationValue">
    </fieldset>

    <fieldset>
      <legend>Log frequency:</legend>
time: Record and log data every n seconds<br>
      n = <input type="number" name="logFrequencyValue">
    </fieldset>
  </form>
</html>
```

```
<fieldset>
  <legend>Directory:</legend>
  location: Save log and images<br>
  <input type="text" name="filepathValeu">
</fieldset>

  <br>
  <input type="submit" id="submitbtn" value="START">
</form>
</html>
```

Appendix D: Configuration File

More recent version may be available at <http://github.com/poxley/4eyes>

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- 4eyes Configuration File -->

<settings>

    <!-- Wait to begin photo sequence -->
    <setting category="delay">
        <type>time</type> <!-- altitude: meters, distance: meters,
time: seconds -->
        <value>30</value>
        <options>
            <option>altitude</option>
            <option>distance</option>
            <option>time</option>
        </options>
    </setting>

    <!-- When to take a photo -->
    <setting category="trigger">
        <type>time</type> <!-- distance: meters, speed:
meters/second, time: seconds -->
        <value>10</value>
        <options>
            <option>distance</option>
            <option>speed</option>
            <option>time</option>
        </options>
    </setting>

    <!-- How long to keep taking photos -->
    <setting category="termination">
        <type>frames</type> <!-- altitude: meters, frames:
quantity, time: seconds -->
        <value>30</value>
        <options>
            <option>altitude</option>
            <option>frames</option>
            <option>time</option>
        </options>
    </setting>

    <!-- Maximum rotational velocity -->
    <setting category="rotation">
        <type>degrees</type> <!-- degrees/second -->
        <value>30</value>
        <options>
            <option>time</option>
        </options>
    </setting>

    <!-- Log frequency -->
    <setting category="log_frequency">
```

```
        <type>time</type> <!-- seconds -->
        <value>5</value>
        <options>
            <option>time</option>
        </options>
    </setting>

    <!-- Storage location -->
    <!-- Placeholder for future work: -->
    <!-- Field to select file storage location -->

</settings>
```

Appendix E: Device Control

More recent version may be available at <http://github.com/poxley/4eyes>

```
# #!/usr/bin/python

# 4Eyes Control Software: 4eyes.py
# Capture NIR+RGB images with Raspberry Pi
# Copyright (c) 2017 Peter Oxley -- poxley@nnu.edu
# Developed for NNU FireMAP and Professor Dale Hamilton M.S.

# Imports
import sys
import time
import datetime
import picamera
import os
import gps
from geopy.distance import vincenty
from Adafruit_BNO055 import BNO055
import RPi.GPIO as gp
import xml.etree.ElementTree as et

#####
# Function: setCam()
# Parameters: cam, the desired camera
# Returns: current camera
# Description: sets GPIO condition
# By: Oxley
#####
def setCam(cam):
    if cam == 2:
        gp.output(CAM_SELECT_PIN, True)
        return 2
    else:
        gp.output(CAM_SELECT_PIN, False)
        return 1

#####
# Function: swapCam()
# Parameters: cam, the current camera
# Returns: (new) current camera
# Description: set the correct GPIO pins to switch cameras
# By: Oxley
#####
def swapCam(cam):
    if cam == 1:
        gp.output(CAM_SELECT_PIN, True)
        return 2
    else:
        gp.output(CAM_SELECT_PIN, False)
        return 1
```

```
#####
# Function: delay()
# Parameters: none
# Returns: continue to delay (T/F)
# Description: determine if user delay requirement has been met
# By: Oxley
#####
def delayStart(stats):
    if DELAY_TYPE == 'distance':
        if (distanceFrom(START_LATITUDE, START_LONGITUDE, stats[2],
stats[3]) < DELAY_VALUE):
            return True
        else:
            return False
    elif DELAY_TYPE == 'altitude':
        if (stats[4] < START_ALTITUDE + DELAY_VALUE):
            return True
        else:
            return False
    elif DELAY_TYPE == 'time':
        if (currentTime() < START_TIME +
datetime.timedelta(seconds=(int(DELAY_VALUE)))):
            return True
        else:
            return False
```

```
#####
# Function: initializeGPS
# Parameters: none
# Returns: start conditions for program constants
# Description: wait until all startup data is available from GPS,
# use to set constants and clock
# By: Oxley
#####
def initializeGPS():
    initFlag = 0
    startAlt, startLat, startLon = 0.0, 0.0, 0.0
    while initFlag < 4:
        if VERBOSE_OUTPUT:
            print(str(currentTime()) + " Waiting for GPS to
initialize")
        initFlag = 0
        gpsReport = gpsSession.next()
        if gpsReport['class'] == 'TPV':
            if hasattr(gpsReport, 'alt'):
                startAlt = gpsReport.alt
                initFlag += 1
            if hasattr(gpsReport, 'lat'):
                startLat = gpsReport.lat
                initFlag += 1
            if hasattr(gpsReport, 'lon'):
                startLon = gpsReport.lon
                initFlag += 1
            if hasattr(gpsReport, 'time') and initFlag == 3:
                os.system("sudo timedatectl set-time " +
gpsReport.time)
```



```

        initFlag += 1
    if initFlag < 4:
        time.sleep(1)
    setupData = (startAlt, startLat, startLon)
    return (setupData)

#####
# Function: terminate()
# Parameters: frame, the current frame sequence number
# Returns: terminate now (T/F)
# Description: determine if capture sequence is complete
# By: Oxley
#####
def terminate(stats, frame, sequenceStartTime):
    if TERMINATION_TYPE == 'altitude':
        if (stats[4] < START_ALTITUDE + TERMINATION_VALUE):
            return True
        else:
            return False
    elif TERMINATION_TYPE == 'frames':
        quantity = TERMINATION_VALUE
        if quantity > frame:
            return True
        else:
            return False
    elif TERMINATION_TYPE == 'time':
        if (sequenceStartTime + TERMINATION_VALUE < currentTime()):
            return True
        else:
            return False

#####
# Function: updateTermFlag()
# Parameters: stats, a list
# Returns: eligible for termination (T/F)
# Description: routine is not eligible for termination until
termination altitude is exceeded
# By: Oxley
#####
def updateTermFlag(stats):
    if TERMINATION_TYPE == 'altitude':
        if (stats[4] > START_ALTITUDE + TERMINATION_VALUE):
            return True
        else:
            return False
    else:
        return True

#####
# Function: captureOne()
# Parameters: frame, the current frame sequence number; camera, the
selected camera
# Returns: none
# Description: capture single image from selected camera

```

```

# By: Oxley
#####
def captureOne(frame, cam):
    prefix = fileName()
    if cam == 1:
        spectrum = 'RGB'
    else:
        spectrum = 'NIR'
    yield (prefix + '%s%03d.jpg' %(spectrum, frame))

#####
# Function: captureImages()
# Parameters: frame, the current frame sequence number
# Returns: frame count
# Description: trigger sub-function (captureOne())
# By: Oxley
#####
def captureImages(frame):
    # cam = setCam(1)
    for j in range(1, 3):
        time.sleep(0.01) # adjust depending on actual latency
        cam = setCam(j)
        time.sleep(0.01) # adjust depending on actual latency
        # for camera.capture_sequence function (below)
        # "yield" command *must* appear within the called function,
        # cannot be called in sub-function.
        camera.capture_sequence(captureOne(frame, cam),
use_video_port=True)
    return

#####
# Function: distanceFrom(lat, lon)
# Parameters: lat and lon: a set of coordinates
# Returns: euclidean distance
# Description: takes lat and lon of some point and calculates distance
from current location
# By: Oxley
#####
def distanceFrom(startLat, startLon, endLat, endLon):
    here = (endLat, endLon)
    there = (startLat, startLon)
    return abs(vicenty(here, there).meters)

#####
# Function: fileName()
#####
def fileName():
    return ('{:y%m%d_ %H%M}'.format(datetime.datetime.now()))

#####
# Function: getStats()
# Parameters: none
# Returns: list of stats from GPS and AOB

```

```

# Description: poll GPS and AOB, parse results for desired data, return
data as a list
# By: Oxley
#####
def getStats():
    curTime, curLat, curLon, curAlt, curAgl, curClimb, curSpeed,
curSats, curEpx, curEpy, curEpv, curTrack, curHead, curRoll, curPitch,
currAngVel = "na", "na", "na", "na", "na", "na", "na", "na", "na",
"na", "na", "na", "na", "na", "na", "na"
    curImg = 'n'
    statsReport = gpsSession.next()
    if statsReport['class'] == 'TPV':
        if hasattr(statsReport, 'time'):
            curTime = statsReport.time
        if hasattr(statsReport, 'lat'):
            curLat = statsReport.lat
        if hasattr(statsReport, 'lon'):
            curLon = statsReport.lon
        if hasattr(statsReport, 'alt'):
            curAlt = statsReport.alt
        if hasattr(statsReport, 'climb'):
            curClimb = statsReport.climb
        if hasattr(statsReport, 'speed'):
            curSpeed = statsReport.speed
        if hasattr(statsReport, 'epx'):
            curEpx = statsReport.epx
        if hasattr(statsReport, 'epy'):
            curEpy = statsReport.epy
        if hasattr(statsReport, 'epv'):
            curEpv = statsReport.epv
        if hasattr(statsReport, 'track'):
            curTrack = statsReport.track
    if statsReport['class'] == 'SKY':
        if hasattr(statsReport, 'satellites'):
            rpvtSatellites = statsReport.satellites
            satCount = 0
            curSats = 0
            for sat in rpvtSatellites:
                satCount += 1
                if sat.used:
                    curSats += 1
    curHead, curRoll, curPitch = bno.read_euler()
    xVel, yVel, zVel = bno.read_gyroscope()
    curAngVel = max(xVel, yVel, zVel)
    if curAlt != 'na':
        curAgl = curAlt - START_ALTITUDE
    stats = [curTime, curImg, curLat, curLon, curAlt, curAgl, curClimb,
curSpeed, curSats, curEpx, curEpy, curEpv, curTrack, curHead, curRoll,
curPitch, curAngVel]
    return stats

#####
# Function: imageMilestoneReached()
# Parameters: stats (a list), lastMilestone (previous successful
milestone)
# Returns: true or false

```

```

# Description: determine if current conditions indicate images should
be captured
# By: Oxley
#####
def imageMilestoneReached(stats, lastMilestone, recentSpeed):
    if TRIGGER_TYPE == 'distance':
        startLat, startLon = lastMilestone
        if (distanceFrom(startLat, startLon, stats[2], stats[3]) <
TRIGGER_VALUE):
            return False
        else:
            return True
    elif TRIGGER_TYPE == 'speed':
        if (stats[7] < recentSpeed and stats[7] < TRIGGER_VALUE):
            return True
        else:
            return False
    elif TRIGGER_TYPE == 'time':
        # reformat lastMilestone into datetime
        milestoneDT = datetime.datetime.strptime(lastMilestone, '%Y-%m-
%dT%H:%M:%S.%fz')
        elapsedTime = currentTime() - milestoneDT
        mins, secs = divmod(elapsedTime.seconds, 1000)
        if (secs < TRIGGER_VALUE):
            return False
        else:
            return True

#####
# Function: updateImageMilestone()
# Parameters: stats (a list)
# Returns: new milestone
# Description: provides correct milestone based on image trigger from
.cfg file
# By: Oxley
#####
def updateImageMilestone(stats, oldMilestone):
    newMilestone = oldMilestone
    if TRIGGER_TYPE == 'distance':
        newMilestone = (stats[2], stats[3])
    elif TRIGGER_TYPE == 'speed':
        newMilestone = oldMilestone
    elif TRIGGER_TYPE == 'time':
        newMilestone = (stats[0])
    return newMilestone

#####
# Function: currentTime()
#####
def currentTime():
    return datetime.datetime.now()

#####
# Function: writeLogHeader()

```

```

# Parameters: none
# Returns: none
# Description: write column headers in logfile
# By: Oxley
#####
def writeLogHeader():
    columns = ("{0:>8}, {1:>3}, {2:>13}, {3:>13}, {4:>6}, {5:>6},
{6:>5}, {7:>5}, {8:>4}, {9:>4}, {10:>4}, {11:>4}, {12:>5}, {13:>5},
{14:>4}, {15:>5}, {16:>3}").format(
        "time", "img", "lat", "lon", "alt", "agl", "climb", "speed",
"sats", "epx", "epy", "epv", "track", "head",
        "roll", "pitch", ">v")
    LOG_FILE.write(columns)
    return

#####
# Function: writeLog()
# Parameters: stats (a list)
# Returns: none
# Description: get list of PGS and AOB data and write to file
# By: Oxley
#####
def writeLog(stats):
    columns = (
        "{0:>8}, {1:>3}, {2:>13.8}, {3:>13.8}, {4:>6.2}, {5:>6.2},
{6:>4.2}, {7:>4.2}, {8:>3}, {9:>4.2}, {10:>4.2}, {11:>4.2}, {12:>5.2},
{13:>5.2}, {14:>4.1}, {15:>4.1}, {16:>3.1}").format(
        stats[0], stats[1], stats[2], stats[3], stats[4], stats[5],
stats[6], stats[7], stats[8], stats[9], stats[10], stats[11],
        stats[12], stats[13], stats[14], stats[15], stats[16])
    LOG_FILE.write(columns)
    return

#####
# Function: main()
# Parameters: none
# Returns:
# Description: main function
# By: Oxley
#####
def main():
    writeLogHeader()
    # create a list to handle stats from GPS and AOB
    stats = getStats()
    # Wait for start condition to evaluate true
    while delayStart(stats) is True:
        time.sleep(0.2)

    # initialize cameras
    with picamera.PiCamera() as camera:
        camera.start_preview()
        # set control variables
        terminationEligible = False
        recentSpeed = 0.0
        frame = 1

```

```

        imageMilestone = 0
        logMilestone = START_TIME
        sequenceStartTime = currentTime()

        while (terminationEligible is False or terminate(stats, frame,
sequenceStartTime) is False):
            stats = getStats()
            if imageMilestoneReached(stats, imageMilestone,
recentSpeed) and stats[16] < MAX_ROTATIONAL_VELOCITY:
                captureImages(frame)
                stats[1] = frame
                writeLog(stats)
                imageMilestone = updateImageMilestone(stats,
imageMilestone)
            logMilestone = currentTime()
            frame += 1
            elif (currentTime() > logMilestone +
datetime.timedelta(seconds=LOG_INTERVAL)):
                writeLog(stats)
                logMilestone = currentTime()
            if terminationEligible is False:
                terminationEligible = updateTermFlag(stats)
                recentSpeed = stats[7]

        writeLog(stats)
        return

```

```

##### BEGIN #####
# GPIO setup
CAM_SELECT_PIN = 17
AOB_RESET_PIN = 18
gp.setwarnings(False)
gp.setmode(gp.BCM)
gp.setup(CAM_SELECT_PIN, gp.OUT)
gp.output(CAM_SELECT_PIN, False)
# GPS setup
os.system("sudo gpsd /dev/ttyUSB0 -F /var/run/gpsd.sock")
gpsSession = gps.gps("localhost", "2947")
gpsSession.stream(gps.WATCH_ENABLE | gps.WATCH_NEWSTYLE)
# AOB setup
bno = BNO055.BNO055(serial_port='/dev/ttyAMA0', rst=AOB_RESET_PIN)
if not bno.begin():
    raise RuntimeError('Failed to initialize AOB')
# access config file
tree = et.parse('4eyes.cfg')
settings = tree.getroot()

# CONSTANTS
VERBOSE_OUTPUT = True
# get initial conditions
START_ALTITUDE, START_LATITUDE, START_LONGITUDE = initializeGPS()
START_TIME = currentTime()
DELAY_TYPE = settings[0][0].text # altitude, distance, time
DELAY_VALUE = settings[0][1].text
TRIGGER_TYPE = settings[1][0].text # distance, speed, time
TRIGGER_VALUE = settings[1][1].text

```

```
TERMINATION_TYPE = settings[2][0].text # altitude, frames, time
TERMINATION_VALUE = settings[2][1].text
MAX_ROTATIONAL_VELOCITY = settings[3][1].text # delay imaging if
rotating too fast
LOG_INTERVAL = settings[4][1].text # write log every n seconds
DATA_LOCATION = "" # will want to get this from cfg file user select
LOG_FILE = open(DATA_LOCATION + fileName() + "log.txt", "w")

main()

LOG_FILE.close()
gp.output(11, False)

exit()
```